Genetic Programming and Evolvable Machines
https://doi.org/10.1007/s10710-019-09370-4

GPML: an XML-based standard for the interchange of genetic programming trees

Tiantian Dou[1] · Yuri Kaszubowski Lopes[1,2] · Peter Rockett[1] · Elizabeth A. Hathway[3] · Esmail Saber[3,4]

Received: 12 July 2019 / Revised: 13 November 2019

Abstract

We propose a genetic programming markup language (GPML), an XML-based standard for the interchange of genetic programming trees, and outline the benefits such a format would bring in allowing the deployment of trained genetic programming (GP) models in applications as well as the subsidiary benefit of allowing GP researchers to directly share trained trees. We present a formal definition of this standard and describe details of an implementation. In addition, we present a case study where GPML is used to implement a model predictive controller for the control of a building heating plant.

Keywords Genetic programming · Interchange formats · Extensible markup language · Model predictive control

✉ Peter Rockett
p.rockett@sheffield.ac.uk

Tiantian Dou
tdou1@sheffield.ac.uk

Yuri Kaszubowski Lopes
yurilopes@utfpr.edu.br

Elizabeth A. Hathway
a.hathway@sheffield.ac.uk

Esmail Saber
e.saber@lsbu.ac.uk

[1] Department of Electronic and Electrical Engineering, University of Sheffield, Pitt Street, Sheffield S1 4ET, UK

[2] Present Address: Department of Software Engineering, Federal University of Technology – Paraná, Dois Vizinhos, Paraná, Brazil

[3] Department of Civil and Structural Engineering, University of Sheffield, Mappin Street, Sheffield S1 3JD, UK

[4] Present Address: Civil and Building Services Engineering Division, School of the Built Environment and Architecture, London South Bank University, Borough Road, London SE1 0AA, UK

Published online: 27 November 2019

1 Introduction

Genetic programming (GP) has progressed to a point of maturity where real-world applications are beginning to emerge—see, for example, [7]. Many industrial and related processes require an empirical model of the process, and a significant challenge in this is identifying a suitable model structure [22]. Since searching over the space of all models is typically a non-deterministic polynomial (NP) problem, evolutionary methods, and GP in particular, are a valuable means of finding 'engineering' applicable models. Even though GP is well-suited to identifying near-optimal model structures, in common with other machine learning approaches, a significant amount of fine-tuning, validation, etc. is often required to generate the final model for deployment, although some progress is being made on automated pipelines, for example, [20].

Despite progress on the modelling aspects, one very real challenge remains the widespread deployment of GP in real-world applications. Invariably, trained GP models exist in the form of data structures (trees, lists, etc.) in the memory space of the computer implementing the learning procedure. While perfectly fine in a research setting, for real-world deployment of the trained GP model, this represents a difficulty, as illustrated in Fig. 1. (We use examples here of control applications since that is our principal interest, but identical arguments apply to classification/detection and other systems.)

The conceptually simplest set up is to use the same physical computer for both the training and the control application in which case the in-memory data structure containing the trained GP model can simply be accessed by both the training and control portions of a single program. This organization, however, has two major drawbacks:

- GP training (the left side of Fig. 1) typically requires a workstation, computing cluster or even a graphical processing unit (GPU) system whereas the control application may only require a microcontroller or small single-board computer.
- If the single physical computer hosting both processes in Fig. 1 crashes, the GP model will need to be recreated (retrained) before control can be re-established. This lengthy delay may be completely unacceptable in terms of cost, safety, etc.

Clearly what is required is an intermediate interchange format for the trained GP model, as illustrated in Fig. 2 whereby the results of the training phase are stored in a non-volatile way for repeated reuse, or indeed distribution to a number of identical controllers. This alternative system architecture addresses both the bullet points above:

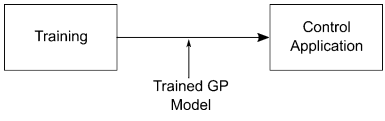

Fig. 1 Implementation of a GP application on the same physical computer

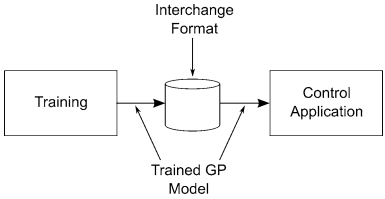

Fig. 2 Implementation of a GP application with an intermediate interchange format; note that the two computational processes no longer have to be implemented on the same physical computer

- The existence of an interchange format means that the GP training and the control application can be easily implemented on physically different, appropriately-sized and even geographically-separated computer systems.
- If the control computer system crashes, the control system can be rebooted, the GP model simply reloaded from the interchange format, and control rapidly re-established.

Having identified the need for a GP interchange format, we turn to a suitable mechanism. Although a custom binary format, perhaps mirroring the in-memory data structure of the trained GP model, would be time and space-efficient,[1] the drawbacks of binary formats are many: different mappings between data types (signed versus unsigned byte-wide character types, different endian-ness of floating-point quantities, etc.), and different file system implementations all present significant problems when the training and control systems are two different computers, or even the same type of computer but using different implementation software for each of the two components. Clearly an easily-transportable, text-based format is mandated. It is worth remarking that, driven by the requirements of inter-operability, most commercial and open-source word processors have abandoned (proprietary) binary formats in favor of open, text-based formats in recent years.

If the interchange format (depicted in Fig. 2) is a human-readable, text-based format, a trained GP model becomes a 'plug-in' component that can be easily improved and/or updated off-line to track, say, process 'drift' and then very quickly deployed in the application system. Furthermore, in software engineering terms, decoupling the training and application functions conforms to recognized good practice.

As to a suitable form for the text-based interchange information, rather than devise a new format, we argue that it makes good sense to adopt one of the established, general-purpose markup languages. These have the benefits of a large user base, standardization and widespread software support across a range of platforms. We consider the requirements and trade-offs in selecting a suitable markup standard in Sect. 2: to pre-empt that discussion, we have based our standard interchange format on the eXtensible Markup Language (XML) [32]. This paper presents what is, as far as the authors are aware, the first definition of a standard, text-based interchange format for GP models. As we have argued above, this provides a key, but

[1] Such as the checkpointing facility in DEAP (https://deap.readthedocs.io/en/master/tutorials/advanced/checkpoint.html) that serializes the Python data structure using the Python `pickle` module.

hitherto missing, component for the wider take-up of genetic programming models in real-world applications.

Section 2 discusses the rationale for our choice of markup language from which we conclude that XML fits our requirements. In Sect. 3, we briefly describe the relevant features of XML and its accompanying validation framework, XML Schemas. We describe an XML representation of a GP tree in Sect. 4, and an example implementation in Sect. 5. In Sect. 6 we describe an application of GPML to embedding a trained GP model in a building control application. Although principally intended to facilitate the deployment of GP in real-world applications, an open interchange format for trained GP models also has the subsidiary benefit of allowing the exchange of GP models between researchers; we discuss this potential further use in Sect. 7. In addition, we develop the material in this paper assuming 'standard GP' [18], that is a genetic programming system based on tree data structures. We discuss the application of GP to other genetic programming formulations in Sect. 7. Section 8 concludes the paper.

2 Choice of markup language

Having motivated the requirement for a text-based interchange format for trained GP models in Sect. 1, we turn now to practical implementation. It would clearly be sensible to adopt an existing serialization standard rather than devise a new format—many potential serialization models exist. For example, one of the reviewers of this paper suggested the Extensible Data Notation (EDN)[2] since this natively represents Lisp s-expressions that were historically used in Koza's early work to represent GP trees. We have not considered EDN because, at time of writing, it is casually rather than formally specified and therefore lacks the standardization we require. Similar comments could be made about the large number of sometimes experimental, competing serialization formats, many of which have technical merits, but currently lack standardization and/or wide-ranging support across a range of software platforms.

Since we judge it important to build on an existing standard with good software support, two main options present: the Extensible Markup Language (XML) and the JavaScript Object Notation (JSON). Both are widely used in web and other technologies.

XML was derived from the earlier Standard Generalized Markup Language (SGML), an ISO-standard document description system. XML meets all our requirements of stability and standardization [32], and of tool support. In addition, it has a large and mature 'eco'-system comprising:

- XML Schema, an XML-compliant validation framework [33]
- XQuery and XPath for database-like queries and node referencing of an XML document—see https://www.w3.org/XML/Query/ and https://www.w3.org/TR/xpath-10/.

[2] https://github.com/edn-format/edn.

- eXtensible Stylesheet Language Transformations (XSLT), a stylesheet system for transforming XML documents to other formats (see https://www.w3.org/standards/xml/transformation)
- Standardized application programming interfaces (APIs), such as the Document Object Model (DOM)—see https://dom.spec.whatwg.org/.

JSON was developed as a 'lightweight' alternative to XML, and subsequently, support analogous to XML's XQuery, XPath and validation schema have been developed. These, however, currently lack the stability and maturity of their XML counterparts. In particular, the JSON Schema that permits the validation of the serialized data is, at time of writing, still at the draft proposal stage and therefore lacks the maturity and stability of XML Schema. A further, minor point is that XML allows comments whereas JSON does not.

Two of the oft-stated advantages of JSON over XML are i) smaller file size and ii) human readability. In a comprehensive series of real-world studies of web applications, however, Lee [14] has shown that the size and processing-speed advantages of JSON are largely unsubstantiated and mostly "myth". As to the superior human readability of JSON, this is a very subjective property, and one with which we disagree. In particular, we consider that for deeply nested structures, JSON's heavy 'overloading' of the three ASCII bracket forms—"()", "{}" and "[]" produces a *less* readable document. On the other hand, we consider that XML's delineation of 'clauses' with unique tags greatly aids readability; it is, of course, this use of more verbose tagging that tends to make XML files slightly larger but highlights a trade-off between readability and terseness. Others may hold a contrary opinion on readability, but human readability is a fairly minor consideration in the present situation.

We should, in passing, also mention the recursively-named YAML Ain't Markup Language (YAML).[3] which can be considered a superset of JSON with arguably superior type derivation. At time of writing, YAML would appear even less mature than JSON.

In summary, after weighing up all factors, we have based our interchange format on XML due to its stability, standardization and maturity, its widespread software support, and the stability/maturity of the accompanying support, particularly the XML Schema validation standard. This design decision, however, is not irrevocable—XML and JSON representations must logically contain identical information [14]. Indeed a number of XML-to-JSON (and vice versa) converters are available.

3 Extensible markup language (XML)

The intrinsic hierarchy in XML lends itself perfectly to describing GP trees, which are, of course, typically represented as acyclic hierarchical graphs. Further, the syntax of XML is very intuitive although human readability is probably a

[3] See https://yaml.org/.

secondary consideration here since we envisage trees being written/read mainly by computer. Nonetheless, the ability to visually inspect the tree structure is valuable, and often requested by paper reviewers. We give some simple examples of GP trees represented with XML in Sect. 4.

In XML, an element is specified by a syntax such as:

```
<elementName option='A'>
    ...
</elementName>
```

where "elementName" and "option" are user-defined identifiers, the latter specifying an attribute, and "A" is a quoted text string. Alternatively, an equivalent single-line version where the element does not nest other elements is:

```
<elementName option='A'/>
```

Crucially for the present application, XML elements can also embed other XML elements, as in:

```
<elementName option='A'>
        <otherElementName...>
        </otherElementName>
        <thirdElementName...>
        </thirdElementName>
        ...
</elementName>
```

and so on to arbitrary levels of nesting. XML can thus straightforwardly represent hierarchical structures. See [8] for a concise but comprehensive introduction to XML.

In a given application, an XML file has to conform to a specified structure—that is, to be *well-formed*. The need for validating XML files has led to the development of XML Schemas [8, 29, 33] for this purpose. XML Schemas—themselves XML-compliant—are able to specify an XML file structure using a syntax reminiscent of the extended Backus Naur format (EBNF) widely used for specifying the grammar of programming languages; importantly for the present application, XML Schemas are able to specify recursive structures for validation.

In terms of implementation, a large number of proprietary and mature open source XML libraries are available, for example, Xerces [1], with bindings to a range of programming languages. In addition, many XML implementations for Matlab are available (e.g. [16]). Consequently, there seems no technical impediment to adopting XML as an interchange format.

We term the interchange format proposed here a Genetic Programming Markup Language (GPML) to denote its GP-specialisation over plain XML.

To date, XML has found little application in the GP community. In a rare example, Tanev and Shimohara [28] have used the Document Object Model

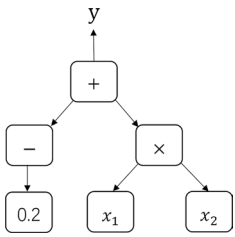

Fig. 3 Simple example GP tree

(DOM) [30], a representation of an XML structure in memory, directly for GP evolution although the DOM is probably not ideal for this purpose.

4 GPML specification

In this section, we describe GPML, a standardised XML-based interchange format.

Making use of the intrinsic hierarchy, GPML recursively encodes the tree structure. Different types of nodes and their corresponding information are identified and recorded in various elements and attributes in GPML. By recursively interpreting the element name and attribute information of each node starting from the root node down, the GP tree can be saved and restored.

With elements and attributes arranged in a nested, hierarchical fashion, GPML can be used to describe GP trees intuitively. We consider a GP tree representing the quite general mapping:

$$\mathbf{x} \overset{f}{\rightarrow} y \tag{1}$$

where co-domain $y \in \mathbb{Y}$ is a set of either: booleans, signed integers, doubles, characters or character strings. The domain $\mathbf{x}$ is an N-tuple drawn from the Cartesian product $\mathbb{L} \times \mathbb{M} \times \ldots$. The sets $\mathbb{L}$, $\mathbb{M}$, etc. are, in turn, sets of either: booleans, signed integers, doubles, characters or character strings. Often, $\mathbf{x} \in \mathbb{R}^N$ when, of course, the input will become a conventional vector of reals.

The nodes in GP trees can be classified as one of a number of types: inputs (i.e. leaf nodes), constant nodes, nodes calling automatically-defined functions (ADFs), unary nodes, binary nodes, ternary nodes, and the more general n-ary node. These different nodes correspond to eight basic elements within GPML, namely, "`input`", "`inputTuple`", "`constant`", "`adfCall`", "`unary`", "`binary`", "`ternary`" and "`nAry`" with user-specified information described in their respective attributes.

For the purpose of initial illustration, consider the simple GP tree shown in Fig. 3 that represents the mapping $y = f(\mathbf{x})$ where $y \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^N$ (although there is no restriction with GPML on the input/output being real numbers, or indeed that the elements of $\mathbf{x}$ are even of the same type). This mapping can be directly represented by the GPML code shown in Listing 1 which incorporates the hierarchy of the GP tree. The whole enclosing structure representing the single instance of the tree ("`gpTree`") has the binary addition ("`" binary operation="+">`") node

as its root. The two children of this root node element are represented as two child elements in GPML: the unary "−" node and the binary " ×" node. Each of these child elements has child elements in turn. In the case of the unary node, its child element is a terminal constant node returning a (default "double") value of 0.2; the two children on the binary " ×" are the terminal "input" nodes returning the first and second elements of the input tuple, respectively.

In Fig. 3, terminal nodes in the GP tree indicate elements of the input vector. We have used the "tupleIndex" attributes of the "input" elements as indices into the input tuple (vector) **x**. For constant nodes, the exact values are specified in the content of the GPML element.

In GP, each unary node has a single child node, which could be any type of node, including another unary node type. In GPML, a unary node is represented by an element "unary", and its operator is specified by its attribute. Similarly, binary, ternary and *n*-ary nodes have two, three or *n* child nodes, respectively. In GPML, node operators are specified by their attributes, like the nodes shown in Fig. 3 and their GPML representations in Listing 1.

Note that each GPML document has exactly one root element, which encloses all the other elements. In GPML, the element name of the XML root node is "gpTree". The "noTupleElements" attribute of the root node defines the number of independent variables used in the mapping, which can be exploited for validation of a GPML document. The indices of terminal nodes in GPML are restricted to the ranges of either $[0 \ldots (\varepsilon\text{noTupleElements}\varepsilon - 1)]$ or $[1 \ldots \varepsilon\text{noTupleElements}\varepsilon]$, depending on the (implementation-defined) convention adopted for indexing tuples in the actual implementation. Consequently, the attribute "firstIndex" $\in$ {'0'|'1'} unambiguously associates tree inputs with elements in the input tuple. The value of "tupleIndex" in a "input" element needs to fall in the appropriate range otherwise a validation error occurs that can be straightforwardly detected and notified to the user. It is also a trivial matter to use a tree that has been trained in a system with zero-index tuples in a (separate) system using tuples indexed from unity, and vice versa.

Listing 1: GPML representation of the tree in Figure 3.

```
<?xml version=''1.0'' ?>
<gpTree noTupleElements=''2'' firstIndex=''1''>
        <binary operation=''+''>
                <unary operation=''-''>
                        <constant> 0.2 </constant>
                </unaryNode>
                <binaryNode operation=''*''>
                        <input tupleIndex=''1''/>
                        <input tupleIndex=''2''/>
                </binary>
        </binary>
</gpTree>
```

4.1 Formal definition of GPML

In terms of extended Backus–Naur form [12] typically used to specify programming languages, we can formally describe the topmost-level "gpTree" element of GPML using:

gp-tree = "<gpTree",
"noTupleElements=", *xml-positive-integer*,
"firstIndex=", "0" | "1" ">",
[*adf-definition-list*],
node,
"</gpTree>";

The terminal symbol "noTupleElements" is an XML-defined primitive of positive integer type [33], i.e. a number $\in \mathbb{N}^+$ denoting the number of elements in the input tuple; the GP mapping is presumed to have at least one input. The non-terminal symbol *xml-positive-integer* value indicates the number of elements in the tuple of input variables for the mapping described by the GP tree. As discussed above, the "firstIndex" attribute denotes whether the described GP tree indexs the input tuple starting from zero or one.

GPML is able to embed an arbitrary number of automatically defined functions (ADFs) [13] as (optional) definitions within the enclosing "gpTree" structure; as shown above using:

adf-definition-list = [*adf-definition*], [*adf-definition-list*] ;

that is, zero or more *adf-definition* types.

A single ADF definition resembles that of a "gpTree" as:

adf-definition = "<adfDefinition", "name=", *adfNameType*, ">"
node,
"</adfDefinition>"

where "name" specifies the textual name of the ADF instance; we have adopted the usual C identifier naming convention of an alphabetical character or underscore followed by any number of alphanumeric characters or underscores. An *adfNameType* is derived from an *xml-token*, an XML string primitive, with the additional restriction that it conforms to the lexical constraints on the ADF's name. Note that the above *adfDefinition* does not allow ADFs to be defined *inside* other ADFs in keeping with a common convention in programming languages.

Note that neither a *gpTree* nor an *adf* is constrained to return a scalar result—either could, for example, return a string or indeed a tuple of quite general form—although the example trees we show in this paper return real values for simplicity of presentation. We reiterate that GPML is concerned solely with representing trained, *semantically-correct* GP trees. GPML is wholly agnostic about how these trees are produced, as should be the case for a general-purpose representation format. To

produce semantically-meaningful trees, mixing data types in the same tree clearly requires an evolution mechanism that is type aware. Discussion of this, however, is out of the scope of the present paper.

The types of scalar data allowable within GPML are defined by:

$$\textit{scalar-type} = \textit{xml-boolean} \mid \textit{xml-integer} \mid \textit{xml-double} \mid \textit{character} \mid \textit{xml-string} \ ;$$

the meaning of most of which are obvious from the XML standard; the *character* type is an *xml-string* that is restricted to a length of one since XML does not define an explicit character type.

A *tuple* is defined recursively as:

$$\begin{aligned}\textit{tuple-type} = {} & \textit{xml-boolean} \mid \textit{xml-integer} \mid \textit{xml-double} \mid \textit{character} \mid \texttt{"`"}, \\ & \textit{xml-string}, \texttt{"'"} \ , [\ \textit{tuple-type}\ ]\ ;\end{aligned}$$

That is, within a tuple, a string literal is demarcated by single quote marks. If it is necessary to include literal single quote marks *within* the string then these must be included in their XML-specified escaped form (as "`'`")—see https://www.w3.org/TR/REC-xml/#syntax. A tuple thus contains one or more *scalar-type* elements, i.e. booleans, signed integers, etc. Note also that tuples can be heterogeneous, namely, they can contain elements of different types.

The non-terminal symbol *node* is defined as:

$$\begin{aligned}\textit{node} = {} & \textit{input-node} \mid \textit{input-tuple-node} \mid \textit{constant-node} \mid \textit{adf-call-node} \mid \textit{unary-node} \mid \\ & \textit{binary-node} \mid \textit{ternary-node}, \mid \textit{n-ary-node}\ ;\end{aligned}$$

where a *node* is one of either: an *input* node, a constant node, an ADF call node, a unary node, a binary node, a ternary node or an *n*-ary node. This set, which we believe currently covers the practical range of useful node types in genetic programming,[4] are formally defined by:

$$\textit{input-node} = \texttt{"<input"}, \texttt{"tupleIndex="}, \textit{xml-non-negative-integer}, \texttt{"/>"};$$

where *xml-non-negative-integer* is again an XML-defined primitive type for an integer quantity ≥ 0 (i.e. $\in \mathbb{N}^0$).

Alternatively, an *input-tuple-node* is a terminal that returns the whole of the input tuple $\mathbf{x}$—see (1)—and is defined as:

$$\textit{input-tuple-node} = \texttt{"<inputTuple"}, \texttt{"/>"};$$

A *constant-node* is a terminal node that can return either a *scalar-type* or a *tuple-type*. Hence, we define the type of the data returned by a *constant-node* as:

[4] Like any standard, we naturally envisage GPML evolving with time to suit new, changed circumstances.

data-type = "boolean" | "integer" | "double" | "char" | "string" | "tuple" ;

and a *constant-node* is defined as:

constant-node = "<constant", ["dataType=", *data-type*] , ">"
value ,
"</constant>";

where *data-type* is defined above, and specifies the returned type of a constant node. Specifying a *data-type* is optional (as denoted by enclosure in square brackets in the EBNF statement above) and defaults to "double" if omitted since this is probably the most common use case. The *value* field represents the constant value in type-appropriate form. e.g. "true" for "boolean", etc. If the constant node returns a tuple, this is represented as a whitespace-separated list of elements; string tuple elements are demarcated by single quote marks.

An *adf-call-node* is a form of terminal defined by:

adf-call-node = "<adfCallNode", "name=", *xml-token*, "/>"

and can occur in a tree anywhere an "input", "inputTuple" or "constant" node can occur.

Consistent with the naming convention set out above for an ADF, "name" is a string starting with an underscore or alphabetic character followed by any number of alphanumeric characters or underscores.

A unary node is defined by EBNF as:

unary-node = "<unary", "operation=", *xml-token*,
["parameterString=", *xml-string*,]
">",
node,
"</unary>";

in which *operation* is an XML-defined token type (i.e. a character string), and *parameterString* is an optional *xml-string* type. The "operation" element indicates the (implementation dependent) operation executed by the unary node; for a unary minus, for example, this might be "−", or for an exponential function "exp".

For the cases where a unary node may take some (arbitrary number of additional) parameters, these can be specified using the (optional) '"parameterString"' by concatenating all the function's parameters in, say, a comma or whitespace-separated list. These parameters can then be simply 'unpacked' by the implementation code. This method of passing any additional parameters is a carefully considered design

decision for GPML, striking a balance between simplicity, generality and clarity of GPML syntax. For example, the GP induction of a decision tree [4] would typically contain (only) binary nodes that test the state of an element in the input vector and follow the left or right child subtrees, respectively depending on whether the given input element was $<$ than or $\geq$ than some decision threshold. For this application, one possible implementation would be for a node's threshold value to be passed in the optional parameter attribute and extracted by the implementation code.

A range of quotient-type binary operators are commonly used in GP: at least two variants of protected division, analytic quotient [19], and unprotected division. Each can be uniquely distinguished by an (implementation-dependent) *operation* field, and indeed could easily be mixed in the same tree, if desired.

The *binary-node*, *ternary-node*, and *n-ary-node* types in GPML are defined by:

binary-node = "<binary", "operation=", *xml-token*,
["parameterString=", *xml-string*,]
">",
node,
node,
"</binary>";

ternary-node = "<ternary", "operation=", *xml-token*,
["parameterString=", *xml-string*,]
">",
node,
node,
node,
"</ternary>";

n-ary-node = "<nAry", "operation=", *xml-token*,
["parameterString=", *xml-string*,]
">",
node-list,
"</nAry>";

where a *node-list* is sequence of one or more *node* types recursively defined by:

$$\textit{node-list} = \textit{node},\ [\textit{node-list}];$$

One potential use of an *n*-ary node type is to represent the iterative structures that have been previously employed in GP [25]. For example, considering a prototypical `for`-loop:

`for(i = 0;i < 10;i + +){s}`and defining a 4-ary node, the first child could stipulate the initial value of the index (`i=0`), the second child the termination criterion on the index ($\texttt{i} \geq \texttt{10}$), the third the increment on `i`, and the fourth child the sequence to be executed (`s`). Except, of course, in a GP environment, each of these values could be generated by child sub-trees.

Yet again, *operation* indicates the operation to be performed by the binary, ternary or *n*-ary nodes, respectively, and the optional *parameter-string* conveys any

additional parameters required. An n-ary node must have at least one child. Rather than define only n-ary nodes with one, two or three children to represent, unary, binary and ternary nodes, respectively, we have retained specific unary, binary and ternary nodes since, in our view, this improves interpretability of the resulting GPML file. Their use, however, is optional—either method of specifying a binary node, say, could be used.

Furthermore, ADFs are sometimes required to take arguments. Rather than introduce new *argument* terminals that would need to have multiple types and that could only appear in ADFs, we concluded that it would be simpler to exploit an (implementation-dependent) n-ary node where the first sub-tree is a call to an ADF, and the remaining $(n-1)$ subtrees evaluate the arguments that are then passed to the ADF in the normal way and form the values to be used as that ADF's terminal variables. So when such an n-ary (function) node is evaluated, it first evaluates the argument subtrees, passes the argument list to the ADF, and finally returns the result to the parent of the n-ary node. This mechanism then looks rather like the conventional representation of a function as $f(\mathbf{x})$.

The operations specified by the above internal node types are deliberately left undefined by GPML, and are implementation-dependent. Typically, in a regression problem, the unary operation will be one of: unary minus, exponential function, sine function, etc. Similarly, the binary operations implemented will be one of: addition, subtraction, multiplication, (protected) division, or analytic quotient [19], while the ternary operation will typically be `if-then-else`. When learning a boolean problem, on the other hand, the only unary operator would typically be the `NOT` operation, and the binary operations would comprise: `AND`, `OR`, `XOR`, etc. This facility means that GPML can be expanded to include arbitrary, domain-specific operations, possibly parameterized via "`parameterString`" elements.

Note also how a "`binary`" node, for example, 'embeds' two *node* types, each of which is defined as being one of a: "`input`", "`inputTuple`", "`constant`", "`adfCall`", "`unary`", (another) "`binary`", "`ternary`" , or "`nAry`" node. The GPML syntax is thus able to recursively define a GP tree of unbounded extent. Similarly, a *gp-tree* 'embeds' a single *node* implying a single root node for the parent tree. Trees defined as ADFs are thus embedded consistently as subroutine-like objects within GPML.

A user is, of course, free to add XML-compliant comments to a GPML file since these will be subsequently ignored by any XML parser.

4.2 An example tree containing an ADF

To further illustrate the utility of GPML, we show in Listing 2 a simple example of a GPML definition of a tree containing a single ADF.

It should be clear that the tree in Listing 2 implements a mapping $\mathbf{x} \rightarrow y$ where $\mathbf{x} \in \mathbb{R}^2$ and $y \in \mathbb{R}$. The GPML listing defines an ADF called "`myADF`" (in lines 3–8) that returns x_1^2. The 'main' program starts at line 11 and calls the ADF at line 12. The function implemented by the whole GP tree is $y = x_1^2 + x_2$. In keeping with

the general philosophy of GPML, we impose no restrictions on how any ADF is generated.

Listing 2: Example GPML representation of a tree containing an ADF.

```
<?xml version=''1.0'' ?>
<gpTree noTupleElements=''2'' firstIndex=''1''>
        <adfDefinition name='myADF'>
                <binary operation=''*''>
                        <input tupleIndex=''1''>
                        <input tupleIndex=''1''>
                </binary>
        </adfDefinition>

        <!-- 'Main' program -->
        <binary operation=''+''>
                <adfCallNode name='myADF'/>
                <input tupleIndex=''2''/>
        </binary>
</gpTree>
```

5 Implementation

Quite deliberately, we do not specify or indeed restrict implementation details for GPML. In this section, we describe our initial implementation as a point of reference.

5.1 Writing GPML

Given a (trained) GP tree in memory, writing a valid GPML file involves a fairly straightforward recursive descent of the tree. With reference to Listing 1, the first task is to output the preamble of the GPML file that comprises the "`gpTree`" information, the "`noTupleElements`" and "`firstIndex`" attributes, and the closing $\varepsilon>\varepsilon$ character. At this stage, the writing procedure recursively descends the tree (in whatever form this has been implemented) and on 'entering' a node, it emits the appropriate GPML element definition ($\varepsilon<$ `input`ε, $\varepsilon<$ `inputTuple`ε, $\varepsilon<$ `constant`ε, $\varepsilon<$ `unary`ε, $\varepsilon<$ `binary`ε, $\varepsilon<$ `ternary`ε or $\varepsilon<$ `nAry`ε). Then:

- If the GP node is a terminal (either an $\varepsilon<$ `input`ε, $\varepsilon<$ `adfCall`ε or $\varepsilon<$ `constant`ε) then it remains only to emit either the "`tupleIndex`" or other attributes and values, and a tag terminating sequence before return from the recursive call.
- An "`inputTuple`" is an empty element and is terminated with $\varepsilon/>\varepsilon$ before return from the recursive call.

- If the current node is a non-terminal GP node, the "operation" attribute field together with the optional "parameterString" field are emitted, and then the appropriate number of further recursive calls made to visit the children of the current node. On return from the last of these recursive calls, the function needs to emit a closing ε< /unary >ε, ε< /binary >ε, ε< /ternary >ε or ε< /nAry >ε field, and then return.

When the chain of recursive calls finally ends, the only remaining task is to emit the ε< /gpTree >ε terminating tag.

If the tree contains ADFs, these need to be emitted after the "gpTree" line and before the first node description of the 'main' program. The use of conventional symbol tables [23] would be one possible implementation mechanism, but this would ultimately depend on the internal organisation of the system that has trained the GP tree.

5.2 Reading GPML

We have implemented our initial GPML system using the lightweight pugixml XML library,[5] largely for simplicity and convenience. The pugixml library reads the specified XML file into memory as a Document Object Model (DOM) [30], a hierarchical structure that can be traversed using functions built into the XML library. Implementation in terms of a DOM is not the only possible approach; the Simple API for XML (SAX) model is equally viable and possibly faster in execution although its use tends to be more involved. Having created a DOM of the tree, it is a straightforward matter to traverse this data structure, creating and linking GP nodes in an implementation-dependent manner. Again, conventional symbol tables [23] provide a possible implementation method in the case where the tree contains ADFs.

5.3 Validating GPML

One of the important elements presented in this work is a well-developed XML Schema for the validation of GPML. The simple pugixml library we have used does not provide validating facilities although these are provided by other XML libraries, for example, Xerces [1]. These more sophisticated libraries tend, however, to be more complicated to use. In practice, a range of other, convenient validation tools are available, for example: the xmllint[6] command-line validator, which is part of the libxml library. Alternatively, the open-source jEdit[7] text editor has an XML plugin that performs validation against a specified schema. Finally, a number of free-to-use XML validators are available online, for example, http://www.utilities-online.info/xsdvalidation/.

[5] http://pugixml.org/.
[6] http://xmlsoft.org/xmllint.html.
[7] http://www.jedit.org/.

In addition to validating the structure set out in Sect. 4, we have exploited a number of features of XML Schema to add checks that:

- The names given to any ADFs are unique and conform to the specified pattern of an alphabetic character or underscore followed by any number of alphanumeric characters or underscores.
- The "`noTupleElements`" attribute is a positive integer.
- The "`firstIndex`" attribute specifying the index of the first tuple element has a value of either 0 or 1.
- The arity of an "`nAryNode`" is at least one.

It is, however, advisable for any implementation of the GPML 'reader' software (Sect. 5.2) to check that:

- The tuple indices specified in all of the "`input`" attributes are within the range of $\geq$ "`firstIndex`" and $<$ "`firstIndex`" + "`noTupleElements`", i.e. that they index a valid tuple element.
- That the value specified in a "`constant`" can be converted to the appropriate data type.

We have made an XML Schema for GPML available under a Gnu Public Licence (GPL-3) on the GitHub repository (https://github.com/pirlite2/gpml-schema).

6 Case study: model predictive control of building heating

In order to demonstrate the value and extensibility of GPML, we describe a case study of using genetic programming to learn the dynamics of a single-zone building, and to implement a model predictive control (MPC) scheme for the heating system to provide comfortable internal conditions.

MPC [3, 15] requires the ability to predict the response of a system at times in the (near) future. To compactly learn the dynamics of most systems, it is usually desirable to construct an autoregressive model to predict the system's output at discrete time steps; an autoregressive model is one in which values of the system output at previous time steps are used as inputs at the current time step thereby compactly encoding the system's'memory' or inertia. A dynamical model predicting, say, internal temperature, y one time step ahead (OSA) can be represented by [24]:

$$\hat{y}_{k+1} = f(\mathbf{x}_k, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_{k-n}, y_k, y_{k-1}, \ldots, y_{k-m}) \tag{2}$$

where k is the current time step, $\mathbf{x}_k, \mathbf{x}_{k-1}, \ldots, \mathbf{x}_{k-n}$ is the set of n delayed (or lagged) input variables, and $y_k, y_{k-1}, \ldots, y_{k-m}$ is the set of m lagged autoregressive outputs. The modeling challenge is to determine the function f along with n and m, the sizes of the lag sets [24], which amounts to a search over a very large space of options. This challenging search problem motivates our use of genetic programming.

To implement dynamical models using GP, it is necessary to introduce time delay operators into the GP function set as was done very successfully by Hinchliffe and Willis [10] in a conventional GP setting. The significant advantage of the formulation in [10] is that the sets of delayed inputs and outputs, the so-called *lag set*, are evolved alongside the functional mapping f rather than having to be specified in advance, as in [9], for example.

In the context of GPML, we can easily introduce unit time delays by defining a new type of the unary node Δ_1 such that given variable y_k measured at the k-th time step, the value of the regressor at time step $(k-1)$ is given by:

$$\Delta_1(y_k) \rightarrow y_{k-1} \tag{3}$$

That is, the unit delay operator Δ_1 returns the value of the same variable at the previous time step. In fact, following the work of Hinchliffe and Willis, we define time delay operators for one, two and three unit delays. Δ_1 is defined in (3) above, $\Delta_2(y_k) \rightarrow y_{k-2}$, and $\Delta_3(y_k) \rightarrow y_{k-3}$. GP evolution is then able concatenate these delay operators to produce delays longer than three time steps, if required. These delay operations can be straightforwardly incorporated in GPML as:

```
<unaryNode operation=''delay1''> ... </unaryNode>
```

i.e. defining a new operation for a time lag of one unit, and so on for lags of two (`"delay2"`) and three (`"delay3"`) units of time. These delayed regressors can, of course, be evaluated from previous outputs of the GP tree.

Given a dynamical model—in the present case implemented with GP—MPC proceeds by explicitly optimizing the set of N future inputs $\mathscr{X} = \{\mathbf{x}_{k+1}, \mathbf{x}_{k+2}, \ldots, \mathbf{x}_{k+N}\}$ over a predetermined prediction horizon N time steps into the future such that the system produces some pre-specified schedule of internal temperature as closely as possible. Note that at time k, (2) predicts only one step ahead, but the temperature prediction $\widehat{y}_{k+2}$ requires the autoregressive input y_{k+1}, which is unknown at time k since at that point it lies in the future. Consequently, y_{k+1} is approximated by $\widehat{y}_{k+1}$, the predicted quantity from the previous time step. Similarly, predicting $\widehat{y}_{k+3}$ utilizes $\widehat{y}_{k+1}$ and $\widehat{y}_{k+2}$, and so on. The requirement on the prediction accuracy of the model is thus quite stringent; in fact, selecting and calibrating the predictive model is widely quoted as consuming 75% of the cost of a conventional MPC project [11]. The details of the training of the GP dynamical models and their performance on MPC fall outside the scope of the present paper and will be published elsewhere.

The quite general simulation framework we have employed is shown in Fig. 4, and is based on the widely-used EnergyPlus building simulator [5] (Block 1) that allows external monitoring and control of its internal states via Functional Mockup Interfaces (FMIs) [2] (Block 2). The training of the GP model, usually termed *system identification* in the control literature, was performed using data obtained by exciting the (simulated) building in so-called open loop, that is, without any feedback control (Blocks 1 and 3). Once trained and validated (Blocks 4 and 5), the GP model was employed in an MPC framework as outlined above (Blocks 1 and 6). It would be extremely inconvenient and cumbersome to embed the GP training and validation procedures within the building simulation and control framework in

Fig. 4. The complexity of the resulting software would be very high and difficult to manage. Further, incorporating non-GP comparator models would add to this complexity. We have thus chosen to modularize the software and employ GPML for the interchange of GP models. The building simulator was configured to independently generate the open-loop excitation data, which was passed to a fairly conventional GP training framework (shown within the dashed box at the bottom right of Fig. 4 (Blocks 3, 4 and 5). The resulting trained and validated dynamical GP model was then emitted as a textual GPML file, and this used as an input to the (separate) MPC program (Block 6). It is thus clear that GPML has facilitated a high degree of software modularity as well as exposing the key predictive dynamical model to independent scrutiny.

An example of of the results of the model predictive control of a simulated single-zone building are shown in Fig. 5. The upper plot is the controlled internal temperature and shows that this quantity is being credibly maintained within a ± 2 C tolerance band during the specified occupied periods of the building. A full, comprehensive account of the GP training procedures as well as complete MPC results will be published elsewhere. The best model presented in GPML form can be found at: https://figshare.com/articles/gpTreeConstantWFInVall_xml/7398797.

7 Discussion

Throughout this paper we have repeatedly talked about "trained GP models" but we have remained deliberately agnostic about how such 'models' might be 'trained'. We have implicitly assumed that the trained GP model is available in memory as a tree data structure—what has previously been termed "standard GP" [18]—although a number of other approaches to GP training have been explored, such as stack-based GP [26] and grammar-guided GP [18] (and references in those two papers). The PushGP system [26], for example, provides an elegant way of incorporating multiple data types within GP and defining semantically-meaningful operations over these types. Thus, ""`INTEGER.=`"" specifies a comparison between integers whereas ""`FLOAT.=`"" signifies a comparison between floats, and so on. Such typed operations can be straightforwardly accommodated by GPML. For example, in a "`binary`" node comparison, the "`operation`" attribute could be specified as (quite literally) "`INTEGER.=`" for integer comparison, and "`FLOAT.=`" for float comparison, and so on. PushGP genotypes, however, are allowed to contain redundant (unused) items as well as embedding control structures. For PushGP programs containing redundant elements but *without* embedded control structures, it is fairly straightforward to transform a PushGP genotype into a conventional tree structure that could be represented using GPML by tracing the sequence of stack operations used in the genotype's evaluation. GPML representation of PushGP genotypes that include control structures is more involved. We have suggested how n-ary nodes could be used to represent looping structures in Sect. 4 although detailed implementation would require further research that is out-of-scope of the present paper.

Grammar-guided GP [18] typically uses genotypes either in the form of trees or linear lists that are then transformed to a phenotype tree for evaluation. In the

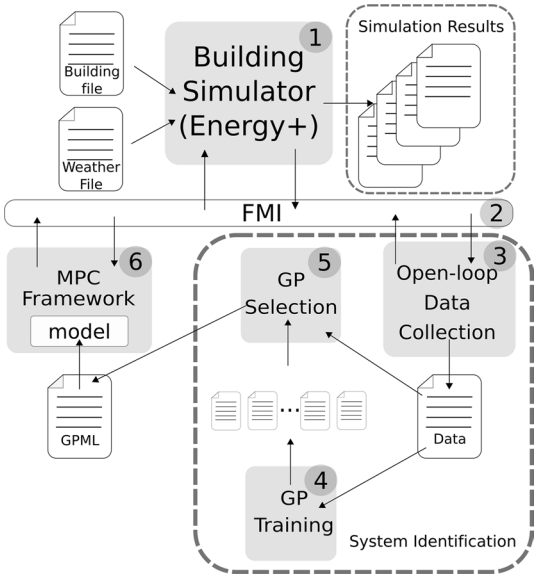

Fig. 4 Overview of the process employed in this work for MPC

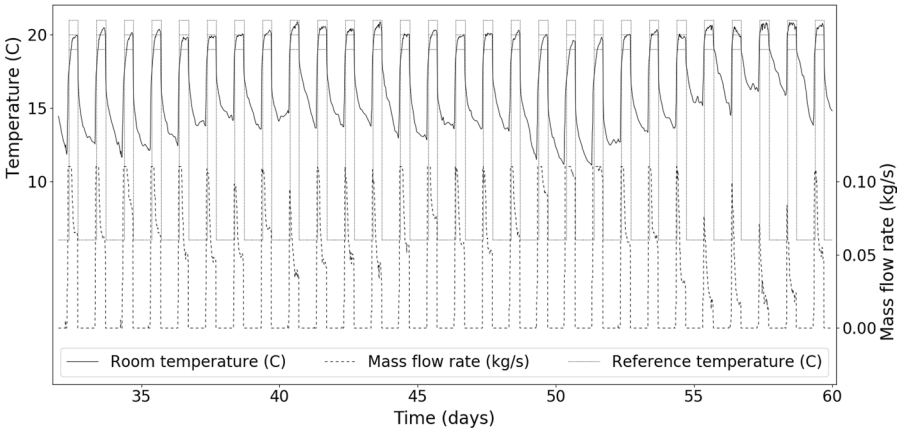

Fig. 5 Typical results of MPC controlling a single-zone building for the (test) month of February. The upper plot is the internal zone temperature, and the lower plot the controlled variable of mass flow rate of water through a radiator. The rectangular upper plot represents the temperature schedule along with a ±2 C comfort tolerance band

context of GP interchange (Fig. 2), it is the phenotype tree that is of interest here and which can be straightforwardly encoded in GPML.

More generally—and anticipating future GP developments—we can consider GP as a quite general deterministic mapping (a composition of functions) that maps elements in Cartesian product spaces to other sets of Cartesian product spaces. In the sense that a graph can represent transitions, it would seem that any semantically-meaningful, deterministic mapping can be represented with a tree. (This does not,

however, imply any judgment on whether genotype representations of non-tree form may be advantageous for *learning*.) Since GPML is able to represent hierarchical tree structures, it would thus appear able to represent any GP phenotype.

Although we have proposed GPML with the principal motivation of moving genetic programming into real-world applications, it also has the potential subsidiary use of allowing researchers to share trained models amongst themselves for the purposes of direct comparison as opposed to individual researchers trying to reproduce each others' work. Extending this further, since journals, and an increasing number of conferences, provide repositories for supplementary material, it would seem straightforward for authors to deposit, and thereby archive, information fully describing *actual* GP trees that could be accessed and used by other researchers. At the suggestion of a reviewer, we also need to explicitly point out that while GPML faithfully describes the *syntax* of a tree, there is an important distinction to be made between syntax and semantics. Unless *operations* are identically defined on both training and target machines, we cannot expect reproducibility.

A great many of the studies published in evolutionary computing, including genetic programming, are—perforce—both empirical and stochastic, and this imposes a particular difficulty from the point of view of replication. In the genetic programming community it has become the universal practice to include algorithm parameters such as population size, crossover and mutation rates, etc. in all publications. Although this is welcome, it generally does not give complete information to allow accurate replication of the experiments being reported. For example, complete reporting would require details of the random number generator algorithms employed, their seed values and other details characterizing the stochastic experiments. It could be argued that an adequate number of repetitions of the experiments 'average out' stochastic effects, but the generality of this claim is far from clear: averages are not always good measures of central tendency. In addition, Demšar [6] has pointed out the possible pitfalls of the null hypothesis statistical tests that are now frequently included in GP papers, and the difficulties inherent in GP comparisons have been discussed in [17]. The ability easily share GP trees may address some of these difficulties.

A similar initiative on standardization has already been taken on GP benchmarking problems [31]. Meanwhile, Orzechowski et al. [21] have advocated standard methods to exchange benchmark results in the GP community. Further afield, Swan et al. [27] have advocated standardized interchange in the metaheuristics community arguing that this would speed research progress. A community-led effort has also recently been announced for the exchange of deep learning models.[8] Although not directly inspired by [27], we see the present work as very much in sympathy to this view.

Finally, as with any standard, there is scope within GPML for revision and expansion. In fact, the reviewers of this paper suggested a number of extensions to GPML that we have implemented with gratifying ease. Further, we suggest that any data structure (e.g. matrices, tensors, quaternions, etc.) that can be represented with XML could be straightforwardly incorporated into GPML should the requirement arise.

[8] https://onnx.ai/.

8 Conclusions

We have proposed an XML-based standard for the interchange of genetic programming trees with the objective of facilitating practical, real-world application of genetic programming. This proposed standard, GPML, allows the interchange in a text-based format of trained-and-validated GP models, in essence treating them as 'plug-in' components. We have demonstrated this capability by developing a dynamical model for a single-zone building model and using this model to successfully implement model predictive control of the building's internal environment. This case study illustrates the practical flexibility of GPML as an interchange format.

In terms of implementation, due to its hierarchical structure, GPML can be flexibly represented in XML for which a number of mature, open source XML libraries are available. We have further proposed an XML Schema for the validation of GPML, which is available under a GPL licence at:

https://github.com/pirlite2/gpml-schema.

As with all standards, we anticipate this definition evolving in the light of practical use and to meet new circumstances. The present authors will be happy to collaborate with the community to evolve this standard.

Acknowledgements This work has been partially supported by the UK Engineering and Physical Sciences Research Council (EPSRC) under Grant EP/N022351/1. We also thank Lee Spector for valuable discussions about PushGP.

References

1. Apache Software Foundation: Apache Xerces Project. http://xerces.apache.org/. Accessed 25 Mar 2017
2. T. Blochwitz, M. Otter, M. Arnold, C. Bausch, H. Elmqvist, A. Junghanns, J. Mauß, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.V.P. an S. Wolf, C. Clauß. The Functional Mockup Interface for tool independent exchange of simulation models. In *8th International Modelica Conference*, Dresden, Germany (2011), pp. 105–114. https://doi.org/10.3384/ecp11063105
3. E.F. Camacho, C. Bordons, *Model Predictive Control*, 2nd edn. (Springer, London, 2004)
4. Y. Cao, P.I. Rockett, The use of vicinal-risk minimization for training decision trees. Appl. Soft Comput. **31**, 185–195 (2015). https://doi.org/10.1016/j.asoc.2015.02.043
5. D.B. Crawley, C.O. Pedersen, L.K. Lawrie, F.C. Winkelmann, EnergyPlus: energy simulation program. ASHRAE J. **42**(4), 49–56 (2000)
6. J. Demšar, On the appropriateness of statistical tests in machine learning. In *3rd Workshop on Evaluation Methods for Machine Learning (ICML 2008)*. Helsinki, Finland (2008)
7. A.H. Gandomi, A.H. Alavi, C. Ryan, (eds.): *Handbook of Genetic Programming Applications* (Springer Nature, Cham, 2015). https://doi.org/10.1007/978-3-319-20883-1
8. K. Goldberg, *XML: Visual QuickStart Guide*, 2nd edn. (Peachpit Press, San Francisco, 2008)
9. B. Grosman, D.R. Lewin, Automated nonlinear model predictive control using genetic programming. Comput. Chem. Eng. **26**(4–5), 631–640 (2002). https://doi.org/10.1016/S0098-1354(01)00780-3

10. M.P. Hinchliffe, M.J. Willis, Dynamic systems modelling using genetic programming. Comput. Chem. Eng. **27**(12), 1841–1854 (2003). https://doi.org/10.1016/j.compchemeng.2003.06.001
11. M.A. Hussain, Review of the applications of neural networks in chemical process control—simulation and online implementation. Artif. Intell. Eng. **13**(1), 55–68 (1999). https://doi.org/10.1016/S0954-1810(98)00011-9
12. International Standards Organization, Information technology—Syntactic metalanguage—Extended BNF (ISO/IEC 14977:1996) (Switzerland, Geneva, 1996)
13. J.R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs* (MIT Press, Cambridge, 1994)
14. D. Lee, Fat markup: trimming the fat markup myth one calorie at a time, in *Balisage: The Markup Conference*. Rockville, MD (2013). https://www.balisage.net/Proceedings/vol10/html/Lee01/BalisageVol10-Lee01.html
15. J.M. Maciejowski, *Predictive Control with Constraints* (Prentice Hall, Harlow, 2002)
16. Mathworks Inc.: XML Documents. http://www.mathworks.com/help/matlab/xml-documents.html. Accessed 25 Mar 2017
17. J. McDermott, D.R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaskowski, K. Krawiec, R. Harper, K. De Jong, U.M. O'Reilly, Genetic programming needs better benchmarks. In *Genetic and Evolutionary Computation Conference (GECCO 2012)*. Philadelphia, PA (2012), pp. 791–798. https://doi.org/10.1145/2330163.2330273
18. R.I. McKay, N.X. Hoai, P.A. Whigham, Y. Shan, M. O'Neill, Grammar-based genetic programming: a survey. Genet. Program Evolvable Mach. **11**(3–4), 365–396 (2010). https://doi.org/10.1007/s10710-010-9109-y
19. J. Ni, R.H. Drieberg, P.I. Rockett, The use of an analytic quotient operator in genetic programming. IEEE Trans. Evol. Comput. **17**(1), 146–152 (2013). https://doi.org/10.1109/TEVC.2012.2195319
20. R.S. Olson, J.H. Moore, TPOT: A tree-based pipeline optimization tool for automating machine learning. In F. Hutter, L. Kotthoff, J. Vanschoren (eds.) *Workshop on Automatic Machine Learning*, vol. 64, pp. 66–74. New York, NY (2016). http://proceedings.mlr.press/v64/olson_tpot_2016.pdf
21. P. Orzechowski, W. La Cava, J.H. Moore, Where are we now? A large benchmark study of recent symbolic regression methods. In *Genetic and Evolutionary Computation Conference (GECCO'18)*. Kyoto, Japan, pp. 1183–1190 (2018). https://doi.org/10.1145/3205455.3205539
22. R.K. Pearson, *Discrete-time Dynamic Models. Topics in Chemical Engineering* (Oxford University Press, Oxford, 1999)
23. R. Sedgewick, *Algorithms in C, Parts 1–4 : Fundamentals Data Structures Sorting Searching*, 3rd edn. (Addison-Wesley, Harlow, 2001)
24. J. Sjöberg, Q. Zhang, L. Ljung, A. Benveniste, B. Delyon, P.Y. Glorennec, H. Hjalmarsson, A. Juditsky, Nonlinear black-box modeling in system identification: a unified overview. Automatica **31**(12), 1691–1724 (1995). https://doi.org/10.1016/0005-1098(95)00120-8
25. L. Spector, J. Klein, M. Keijzer, The Push3 execution stack and the evolution of control, in *Genetic and Evolutionary Computation Conference*. Washington DC (2005), pp. 1689–1696. https://doi.org/10.1145/1068009.1068292
26. L. Spector, A.J. Robinson, Genetic programming and autoconstructive evolution with the push programming language. Genet. Program Evolvable Mach. **3**(1), 7–40 (2002). https://doi.org/10.1023/A:1014538503543
27. J. Swan, S. Adriaensen, M. Bishr, E.K. Burke, J.A. Clark, P.D. Causmaecker, J. Durillo, K. Hammond, E. Hart, C.G. Johnson, Z.A. Kocsis, B. Kovitz, K. Krawiec, S. Martin, J.J. Merelo, L.L. Minku, E. Özcan, G.L. Pappa, E. Pesch, P. García Sánchez, A. Schaerf, K. Sim, J.E. Smith, T. Stützle, S. Voß, S. Wagner, X. Yao, A research agenda for metaheuristic standardization. In *11th Metaheuristics International Conference*. Agadir, Morocco (2014)
28. I. Tanev, K. Shimohara, XML-based genetic programming framework: design philosophy, implementation, and applications. Artif. Life Robot. **15**(4), 376–380 (2010)
29. P. Walmsley, *Definitive XML Schema*, 2nd edn. (Prentice Hall, Upper Saddle River, 2013)
30. WHATWG: Document Object Model (DOM) (2018). https://dom.spec.whatwg.org/. Accessed 25 June 2018
31. D.R. White, J. McDermott, M. Castelli, L. Manzoni, B.W. Goldman, G. Kronberger, W. Jaśkowski, U.M. O'Reilly, S. Luke, Better GP benchmarks: community survey results and proposals. Genet. Program Evolvable Mach. **14**(1), 3–29 (2013)
32. World Wide Web Consortium: Extensible markup language (XML) 1.0 (2008). https://www.w3.org/TR/2008/REC-xml-20081126/. Accessed 17 July 2018

33. World Wide Web Consortium: XML Schemas (2012). https://www.w3.org/standards/xml/schema. Accessed 17 July 2018